Optimizing ASP.NET with C++ AMP on the GPU

High-Performance Parallel Code in the AWS Cloud

Scott Zimmerman

April 2015

Notices

This document is provided for informational purposes only. It represents AWS's current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS's products or services, each of which is provided "as is" without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Licensed under the Apache License, Version 2.0 (the "License"). You may not use this file except in compliance with the License. A copy of the License is located at http://aws.amazon.com/apache2.0/ or in the "license" file accompanying this file. This code is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Portions of the code were developed by Heaton Research and are licensed under the Apache License, Version 2.0, available here: https://www.apache.org/licenses/LICENSE-2.0.html

Portions of the code were developed by Microsoft Corporation and are licensed under Microsoft MSPL, available here: http://opensource.org/licenses/ms-pl

Contents

Abstract

This whitepaper is intended for Microsoft Windows developers who are considering writing high-performance parallel code in Amazon Web Services (AWS) using the Microsoft C++ Accelerated Massive Parallelism (C++ AMP) library. This paper describes an ASP.NET Model-View-Controller (MVC) web application written in C# that invokes C++ functions running on the graphics processing unit (GPU) for matrix multiplication. Since matrix multiplication is of order N-cubed, multiplying two 1024 x 1024 matrixes requires over one billion multiplications, and is therefore an example of a compute-intensive operation that would be a good candidate for GPU programming. This paper shows how to use AWS Elastic Beanstalk and the AWS Toolkit for Visual Studio to launch a Microsoft Windows Server instance with an NVIDIA GPU in the Amazon Elastic Compute Cloud (Amazon EC2) on AWS.

Introduction

Certain types of parallel algorithms can run hundreds of times faster on a GPU than similar serial algorithms on a CPU. This paper describes matrix multiplication as one example of a parallel algorithm that is suitable for GPU programming. Performance increases of this order are obviously very attractive for certain workloads, but there are several technologies that must be understood and integrated in order to achieve these gains.

First, you'll need a GPU programming language or library. The next section briefly discusses the advantages of Microsoft C++ AMP, and this whitepaper includes working code examples written in C++ AMP. Second, this paper will describe how to use the AWS Toolkit for Visual Studio to launch Amazon EC2 instances with a GPU, connect to them remotely, and install the NVIDIA GPU graphics driver. Third, although the focus here is on C++ programming, we'll need a simple user interface to display results, and it's typically easier to do this in C# than in C++. So this whitepaper shows a small program written in C# that uses ASP.NET MVC to invoke a function written in C++ AMP. Fourth, bringing ASP.NET MVC into the solution means you also need to add the Internet Information Services (IIS) role to Windows Server and deploy the web application. This will be accomplished from inside Visual Studio with the AWS Elastic Beanstalk service. Of course it's not necessary to develop a web front-end

or use C# to take advantage of C++ AMP, but that is a common use case, so this whitepaper covers how to integrate those technologies with C++ and Windows Server running on Amazon EC2.

Figure 1 shows how the ASP.NET MVC architecture spans the physical tiers in this application, and the coding technologies that will be used on each tier. Note that this simple application doesn't include a data tier. Also, the application tier is only a logical concept in this scenario. It is a way of looking at the C# and C++ algorithms as distinct from the web application, even though they run on the CPU or GPU of the same web server virtual machine.

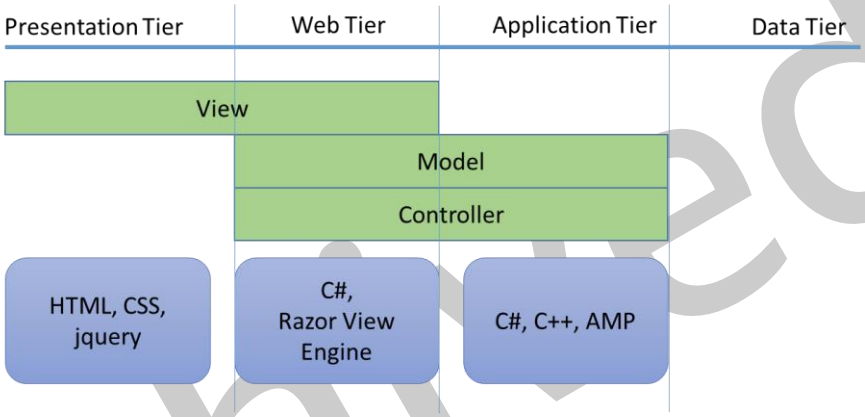

Figure 1: The ASP.NET MVC Architecture and Languages Used

This application starts with a basic matrix multiplication function in C# to show the simplest way to implement the solution. Then the program is optimized six times, each time adding a technology and comparing performance. Subsequent sections of this paper will describe how each variation is coded, and how to set up the technologies.

Download the source code and Visual Studio solution.[1]

Here's an overview of the seven matrix multiplication algorithms that will be illustrated:

Algorithm	Description
C# Basic Serial (CPU)	Written in C# to serve as a performance baseline on which we hope to improve by using C++.
C# Improved Serial (CPU)	Optimizes the order of loop indexes to improve performance.

Algorithm	Description
C# Parallel with TPL (CPU)	Uses the .NET Framework Task Parallel Library (TPL). When run on a machine with multiple cores, this multithreaded algorithm improves performance when compared with the serial C# version.
C++ Basic Serial (CPU)	Converts the basic serial algorithm to C++ to demonstrate how to invoke C++ code from ASP.NET MVC C# code running on IIS.
C++ Parallel with PPL (CPU)	Rewrites the serial C++ function to make it parallel by using the Microsoft Parallel Patterns Library (PPL).
C++ Parallel with AMP (GPU)	Rewrites the parallel C++ function to run on the GPU using basic techniques with C++ AMP.
C++ Parallel with AMP Tiling (GPU)	Rewrites the AMP C++ function to use AMP with tiling. Implementing tiling algorithms takes a bit more work than basic AMP, but if done carefully, it can improve performance.

The performance comparisons illustrated in this application are not meant to be scientific benchmarks, but they may provide useful insight into the potential relative performance of the various techniques. The algorithms are not intended to be optimal. If you really need to do fast matrix multiplications, you should look into tested and optimized libraries such as Basic Linear Algebra Subprograms (BLAS) or Linear Algebra Package (LAPACK).

Introduction to C++ AMP

Until now, programming the GPU has been tedious or non-portable, or limited to the C language. Microsoft C++ AMP enables Visual C++ developers to optimize compute-intensive programs in a highly productive way. AMP is an open specification for an extension to standard C++ that greatly simplifies porting parallel algorithms from the CPU to the GPU. AMP is also elegant and takes advantage of modern C++ features such as lambdas. You'll see that after taking the first step with AMP, parallel code still looks similar to the original serial code.

The popular OpenCL library is portable across multiple operating systems and GPU hardware vendors. It's been around longer than C++ AMP and is recognized for providing very fast run-time performance. However, OpenCL is a C-language library that misses out on modern C++ features.

AMP is portable across GPU hardware, but because it's designed for DirectX, it runs on Windows. In 2012, Intel released a free download called Shevlin Park as a proof-of-concept that enables C++ AMP code to run on top of OpenCL, which means your C++ AMP code can run on Linux and other operating systems.

In 2013, the HSA Foundation published an open-source C++ AMP compiler[2] that outputs OpenCL code. This also enables you to write C++ AMP code to run on Linux and other operating systems.

Microsoft maintains a C++ AMP Algorithms Library modeled after the Standard Template Library[3] and a few dozen C++ AMP sample projects on the AMP blog.[4]

Introduction to Amazon EC2

Amazon EC2 is a service that allows customers to run Windows Server and Linux in the AWS cloud. Amazon EC2 provides over 30 types of compute instances,[5] including memory-optimized, storage-optimized, and GPU-enabled instances. The G2 double extra-large (`g2.2xlarge`) instance type has eight virtual CPUs and an NVIDIA GPU with 1,536 CUDA cores and 4 GB of video memory. CUDA is a parallel computing platform and programming model invented by NVIDIA.[6]

Install the AWS Toolkit for Visual Studio

This paper assumes that you have Visual Studio Professional 2013 or Visual Studio Community 2013 already installed on your computer. It is possible to write the code with Visual Studio Express; however, that edition doesn't support plug-ins such as the AWS Toolkit for Visual Studio. The AWS Toolkit makes it very convenient to perform several account management tasks without ever leaving Visual Studio. You'll use the AWS Toolkit extensively to launch and administer an Amazon EC2 instance in AWS, although it's also possible to do that with the Amazon EC2 console in a web browser.

Please download and install the AWS Toolkit for Visual Studio[7] from the AWS website. For this whitepaper, please ensure you have at least version 1.8.1.0 of the AWS Toolkit for Visual Studio. After installing the toolkit, you should see an option for the AWS Explorer appear in the Visual Studio **View** menu.

Set up the Amazon EC2 Windows Server Instance with NVIDIA GPU

This paper assumes that you have an AWS account with permission to launch Amazon EC2 instances. AWS provides a limited free tier[8] for one year for new customers to experiment with cloud computing. The free tier covers several

services, including Amazon EC2. However, it applies to the T2.micro instance type, not the G2 instance type.

> **Important** Please be aware that there is a cost to run the G2 instance type used in this paper. This profile in the AWS Simple Monthly Calculator shows the estimated cost to run one on-demand G2 instance with Windows Server non-stop for a month. Note that significant cost savings can be achieved by using spot or reserved instances rather than on-demand instances, and by stopping the instance when it's not in use.

The following sections explain how to use the AWS Toolkit for Visual Studio to launch a G2 instance with Windows Server.

Create a Security Group with the AWS Toolkit

Microsoft Remote Desktop Connection (RDC) is useful for manually administering Windows Server remotely, but the NVIDIA display driver that you need for the GPU, and the Remote Desktop Protocol (RDP) used by RDC, are not compatible. RealVNC offers a free version of their VNC Server software that enables remote connections graphically, and it uses a different protocol that is compatible with the NVIDIA driver. So before you install the NVIDIA driver, you will need to install VNC Server on the instance. Then you can disconnect from RDP, reconnect over VNC, and install the NVIDIA driver. Don't worry about installing that now; the detailed instructions are provided later.

RDP uses port 3389. VNC Server uses port 5900. And of course the web application will use port 80. The default security group when launching a Windows Server instance only opens port 3389. You could simply add rules to the default group after you launch the instance, but instead, you'll create your own custom security group and give it a name. You'll also use this custom security group later when you deploy the web application with AWS Elastic Beanstalk.

To create a security group in the AWS Toolkit:

1. In Visual Studio, on the **View** menu, click **AWS Explorer** (or press Ctrl+K, A).
2. Expand **Amazon EC2**, and double-click **Security Groups**. Your security groups are displayed in the right pane. On the menu bar above that pane,

click **Create Security Group**. Fill in the **Name** and **Description**, and leave the **No VPC** option selected, as shown in Figure 2. Click **OK**.

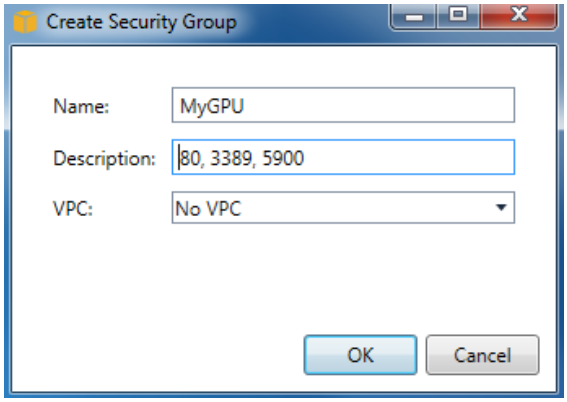

Figure 2: Creating a Security Group

3. Step 2 creates an empty security group. Now let's add the rules to it. In the lower pane, click **Add Permission** to open the **Add IP Permission** dialog box, as shown in Figure 3. Leave **Protocol** as **TCP**. For **Port Range**, type 5900 for both the **Start** and **End** fields. Click **OK**.

Caution For RDP and VNC, it's highly advisable to limit the Source CIDR to your local IP address, with either /32 or an appropriate subnet of your private network appended to the address. You may use the estimated IP address shown in the **Add IP Permission** dialog box (Figure 3), or you can type "what is my IP" into a search engine to see your public IP address. AWS creates a default RDP rule with Source CIDR as 0.0.0.0/0 (which means the whole Internet) to simplify the experience for new users who are launching an instance. But opening VNC and RDP ports to the whole Internet means that hackers can try to guess your administrator password to gain control of your server.

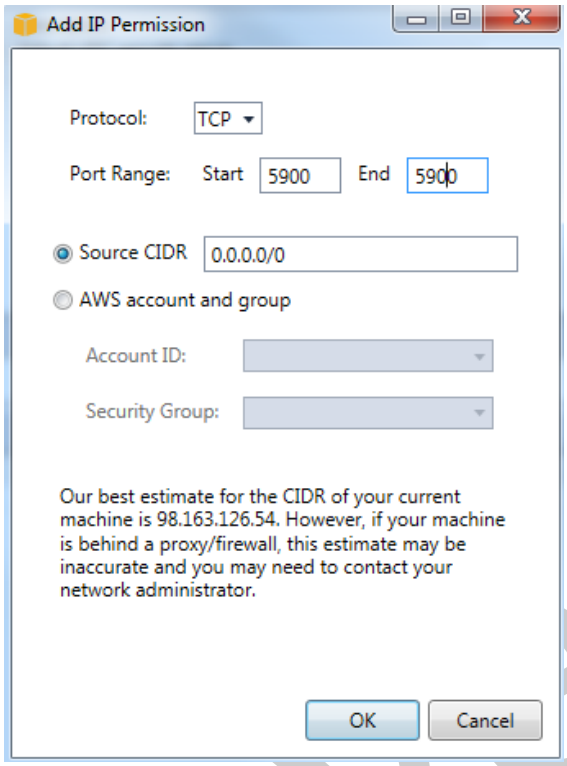

Figure 3: Adding a Rule in the Security Group

4. Repeat step 3 to add port 3389 (for **Protocol**, you can select **RDP**).
5. Repeat step 3 once more to add port 80 (for **Protocol**, you can select **HTTP**). With your security group selected in the top pane, your rules should appear in the middle pane, similar to Figure 4.

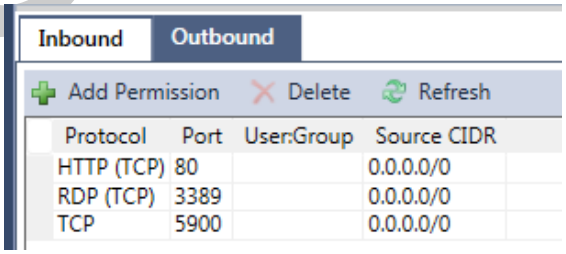

Figure 4: You should Have Three Rules in Your Security Group

Note This security group will serve you while you are installing software on the Amazon EC2 instance. After you complete that task and create an Amazon Machine Image (AMI), AWS Elastic Beanstalk will apply an automatic security group with only ports 22 and 80 open. So if you need to manually administer your Amazon EC2 instance after deploying with AWS Elastic Beanstalk, you must add port 5900 to that security group.

Launch G2 Instance in Amazon EC2 with the AWS Toolkit

Now that you have a custom security group, you're ready to launch a G2 instance:

1. In Visual Studio, on the **View** menu, click **AWS Explorer** (or press Ctrl+K, A). AWS Explorer appears as in Figure 5, where it's shown with the Amazon EC2 service expanded.

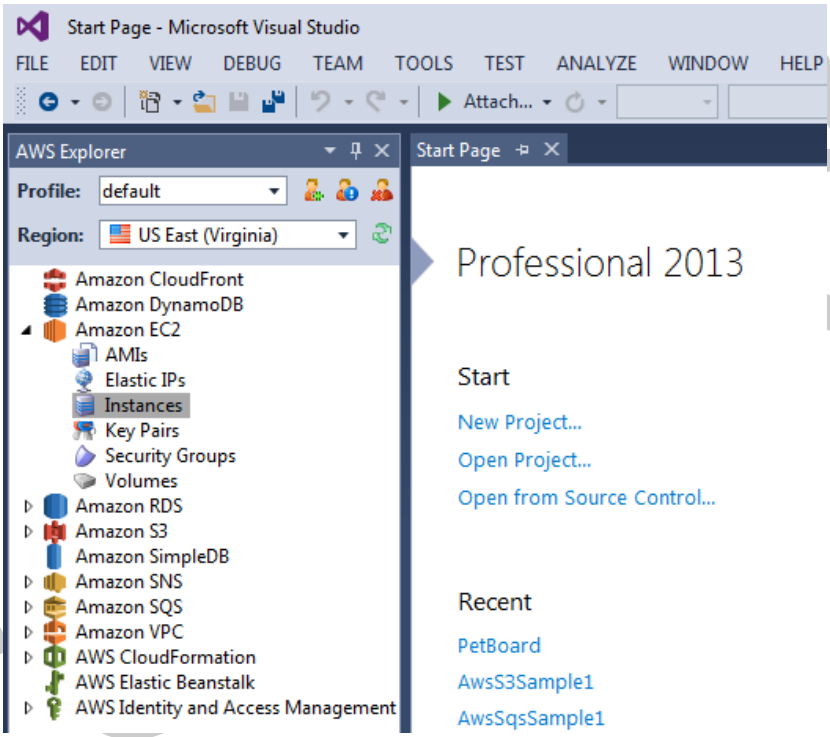

Figure 5: AWS Explorer in AWS Toolkit

2. In AWS Explorer, expand Amazon EC2 as shown in Figure 5. Right-click **Instance**, and then click **New Instance**.
3. In the Quick Launch wizard, click **Advanced**. AWS has created special AMIs to optimize the deployment time for IIS and the .NET Framework with AWS Elastic Beanstalk. The wizard lets you pick one of those AMIs as your base image. After you get your instance prepared with the NVIDIA drivers, you'll save your own AMI.
4. In the **Launch new Amazon EC2 Instance** dialog box (see Figure 6), type **.net beanstalk** in the search text box (the third **Viewing** field). Then change the setting of the first field from **Owned by me** to **Amazon Images**. Do it in that order; otherwise, it takes longer. Click the **Name**

column heading to sort the AMIs by name. Expand the **Description** column so you can see the dates the images were created. Scroll down to select the most recently created Windows Server 2012 R2 (not core) image. At the time this screenshot was taken, the latest version of the Beanstalk Container was v2.0.2.6. However, new images are released from time to time to incorporate the latest Windows updates from Microsoft, so you'lll likely see a newer version. Now click **Next**.

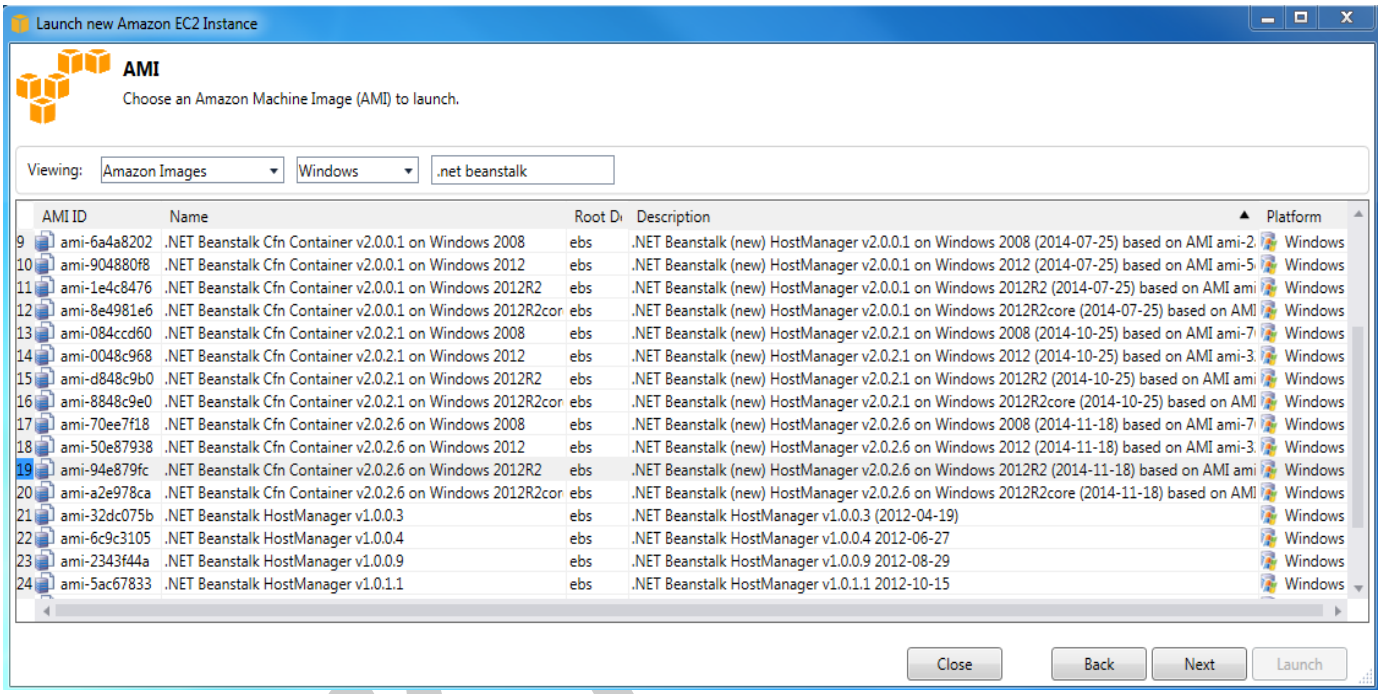

Figure 6: Choosing an AMI

5. In the **AMI Options** dialog box, in the **Instance Type** list, select **GPU Double Extra Large**. Click **Next**.
6. In the **Storage** dialog box, click **Next**.
7. In the **Tags** dialog box, provide a name for the instance so it's easy to distinguish it.
8. In the **Security** dialog box (Figure 7), click **Create New Key Pair**, and give it a name. Choose the security group you created earlier (this is very important).

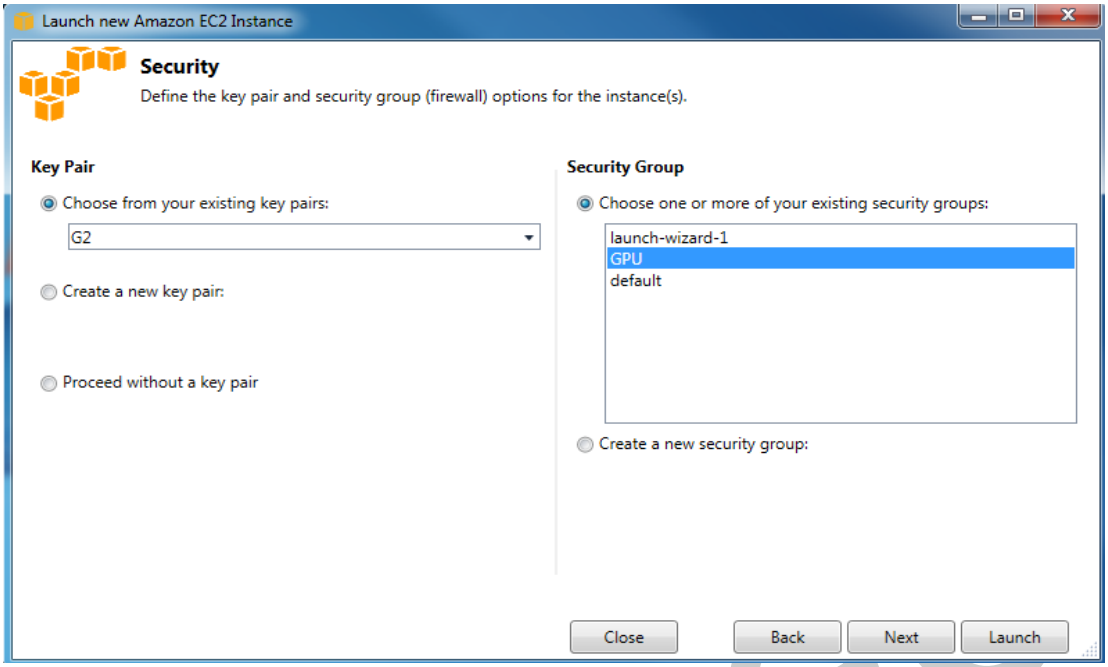

Figure 7: Choosing the Security Group You Created Earlier

9. Click **Launch**.
10. In the AWS Explorer left pane, under **Amazon EC2**, double-click **Instances**. That will display the panel of your instances, and you should see that your new instance is launching. The status will show as "pending" for a few minutes, and then it will change to "running." You can continue to the next step while the launch is pending.
11. You'll need an Elastic IP address for this instance so you can easily reconnect to it if you stop and restart the instance. Right-click the instance (even if the status is pending), and then click **Associate Elastic IP**. In the **Attach Elastic IP to Instance** dialog box (Figure 8), click **Create new Elastic IP**, and then click **OK**.

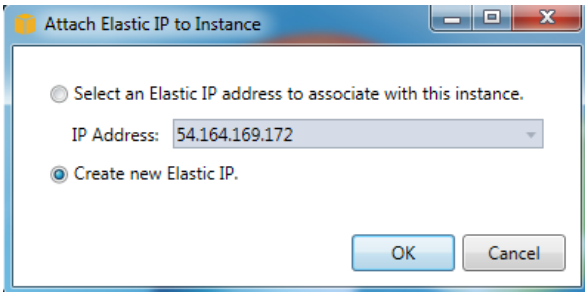

Figure 8: Creating a New Elastic IP

Note Remember, there's an hourly cost for the instance while it's running, so it's a good idea to stop (not terminate) the instance and restart it if you're not able to finish all the steps in this whitepaper in one session.

Connect to the Instance to Install the NVIDIA Driver and Visual C++ Redistributable

In this section, you'll download and install VNC Server on the instance using Microsoft Internet Explorer. But before you can do that, you'll need to turn off the Internet download protection feature that is enabled by default in Internet Explorer 11 on Windows Server 2012 R2.

While you're on the instance, you'll also download and install the Visual C++ 2013 redistributable package. Doing this manually is simpler than creating a setup program with a merge module. The reason you'll do this now is so you can create a fully prepared AMI of the instance that you can use later to deploy your web application with AWS Elastic Beanstalk.

For some of the steps in this section, you'll use the AWS Toolkit on your local workstation; for others, you'll use the Amazon EC2 instance connected through RDC or VNC. The transitions will be mentioned as needed.

After the status of your instance changes from "pending" to "running," follow these steps in the AWS Toolkit:

1. The AWS Toolkit has a convenient option to log in directly with the key pair we created previously without requiring you to enter the administrator password. This works until you change the password on the instance, which you'll need to do to connect with VNC. Right-click the instance in the AWS Toolkit, and then click **Open Remote Desktop**. In the **Open Remote Desktop** dialog box (Figure 9), leave the **Use EC2 keypair to log on** option selected, and then click OK. The toolkit automatically decrypts the AWS-generated password from the key pair, passes it to Microsoft RDC, launches RDC, and then logs you into the Amazon EC2 instance.

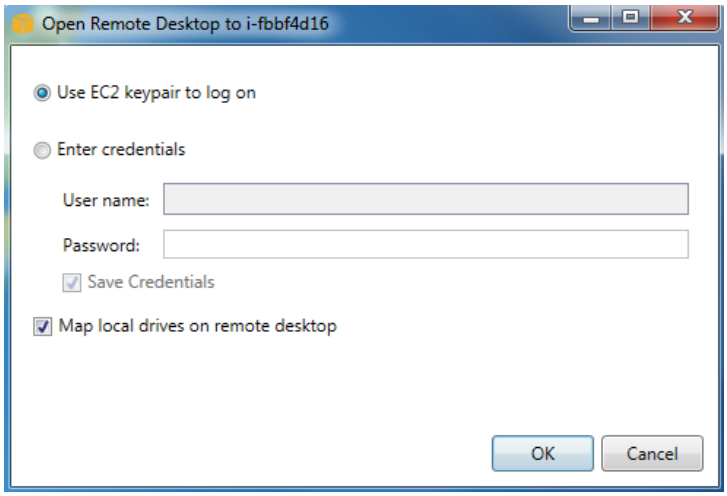

Figure 9: Open Remote Desktop

For steps 2-7, you'll use RDC connected to the instance.

> During steps 2-12, if you get a popup message indicating that Windows has updates to install on the instance, you should go ahead and apply those so they'll be included in the AMI you'll create in step 14. If Windows Update requires a reboot, restart your machine and then resume these instructions after reconnecting through RDC (or VNC Viewer).

2. You must first change the Windows administrator password on the instance to a password you can remember. In Windows Server 2012 R2, click the Windows icon (Start button) in the lower-left corner of your screen to get to the Start menu. Click **Administrative Tools**. Double click **Computer Management**. Expand **Local Users and Groups**. Click once on **Users**. Right-click **Administrator**, and then click **Set Password**. Click **Proceed**. Enter the new password, and then click **OK**. Now the AWS-generated password is obsolete. Close **Computer Management**.

3. To enable file downloads in Internet Explorer, click the Windows Start button again. Click **Server Manager**. In the left pane, click **Local Server**. You should see that Internet Explorer enhanced security configuration is turned on by default. Click to turn it off for administrators, and then click **OK**. Close Server Manager.

4. To run Visual C++ code, you'll need to install the Visual C++ 2013 redistributable from Microsoft. It includes the C++ runtime and the AMP DLL file. Click the Windows Start button again. Click **Internet Explorer**. Browse to the Microsoft download page for Visual C++ Redistributable

Packages for Visual Studio 2013.[9] Click **Download**, and choose the file **vcredist_x64.exe** from the list of downloads. Run the program after downloading it.

5. Open Internet Explorer. Browse to the RealVNC website.[10] Download VNC Server for Windows. The free version is adequate for this whitepaper, but you will need to register with RealVNC to get a license. Install VNC Server (you don't need to install the Printer Driver or VNC Viewer).
6. On the Windows Start menu, click **All Programs** to display all installed applications. Under VNC, click **Enter VNC Server License Key**. Go through the VNC wizard to license your server software.
7. Now you can close RDC, but leave the instance running.

> Now that you will no longer be using RDP with the instance, we recommend that you delete the security group rule that permits RDP traffic to the instance. You still need to leave port 5900 open for VNC.

8. Install and launch the VNC Viewer program on your local workstation.[11] It prompts you for the VNC Server public IP address. To retrieve the IP address, right-click your instance in AWS Explorer, and then click **Properties**. In the **Properties** dialog box (Figure 10), right-click the Elastic IP value, and then click **Copy**. Paste the address into the VNC Server address box in VNC Viewer.

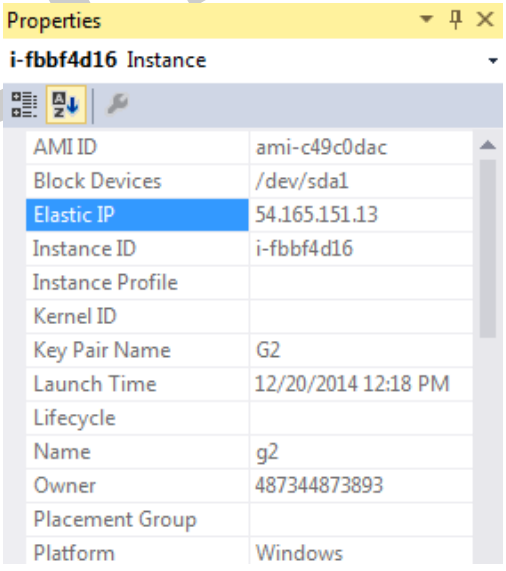

Figure 10: Getting the Elastic IP Address from the Instance Properties

9. When you connect to the instance in VNC Viewer, it will prompt you to press Ctrl+Alt+Delete to log in. Ordinarily, that keystroke sequence is captured by your local workstation. The trick is to slide your mouse toward the top center of the VNC Viewer window. That will drop down the toolbar where you can click the Ctrl+Alt+Delete button to transmit the keystroke to the remote machine. VNC Viewer shows the remote machine prompting you for your Windows administrator password. Type the password that you set in Windows when you logged in previously with RDC.

Do steps 10-12 on the instance while connected through VNC.

10. Open Internet Explorer to download the NVIDIA graphics driver. As of this writing, the latest version on the NVIDIA support site is [NVIDIA GRID K520/K340 Release 334](#)[12] (Figure 11). Although the page title says 334, the version is 335. Regardless, you should be fine if you get the latest version.

When the NVIDIA installation completes, it prompts you to reboot. You can save time if you complete the next few steps first.

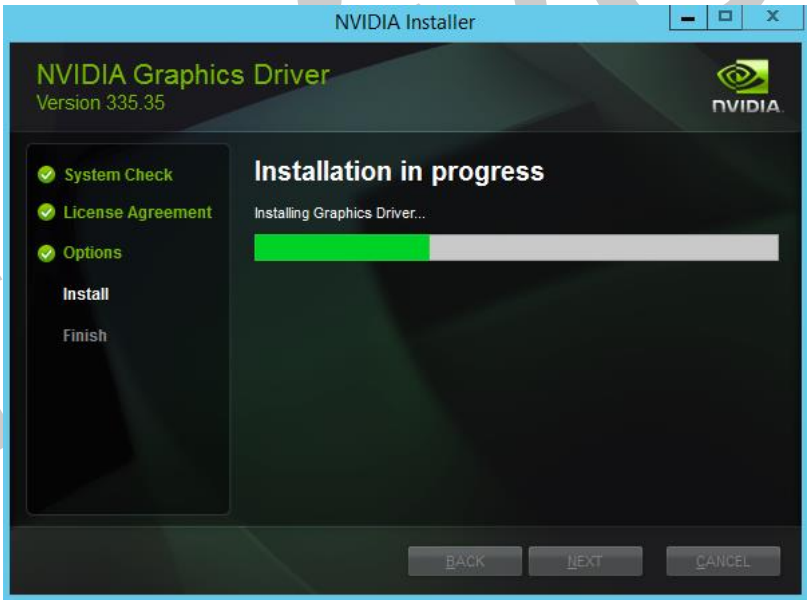

Figure 11: Installing the NVIDIA Graphics Driver

11. Don't reboot after installing the NVIDIA graphics driver. Instead, on the Windows Start screen, type **ec2**, and click to run the **EC2Config service**. To make the image compatible with AWS Elastic Beanstalk, select the **User Data** box on the **General** tab (Figure 12), and choose **Random** for the **Administrator Password** on the **Image** tab (Figure 13). Click **Apply**, and then click **OK**.

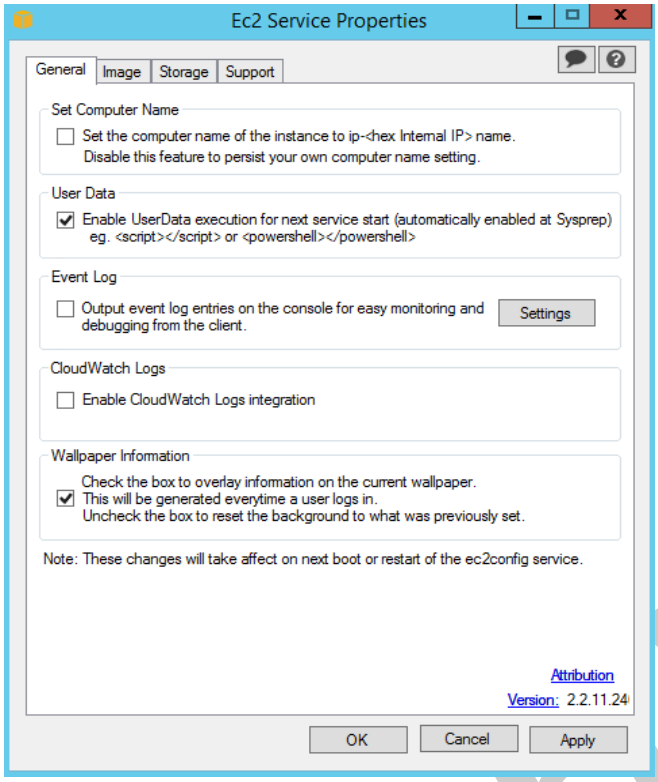

Figure 12: Checking User Data in EC2Config

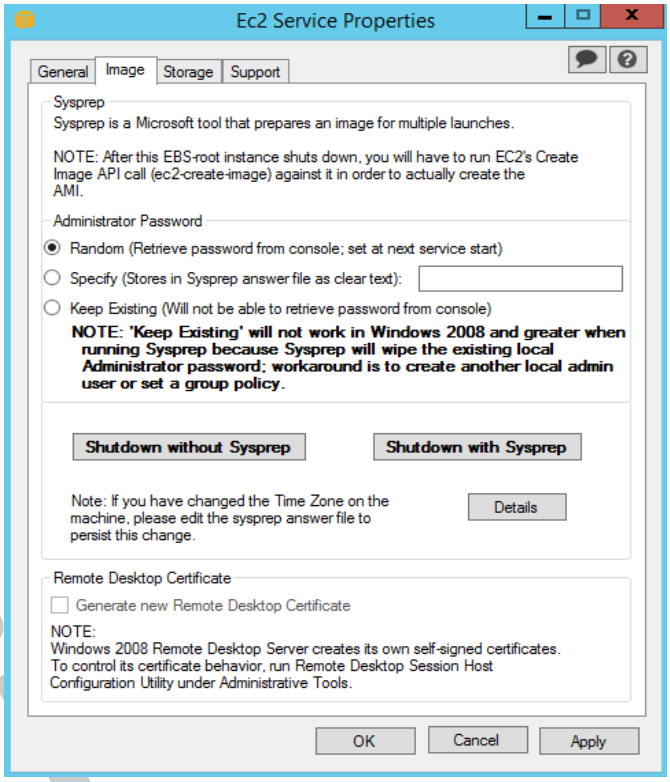

Figure 13: Checking Random Password in EC2Config

12. Click the Windows Start button. Click **Administrative Tools**. Double-click **Computer Management**. Click **Device Manager**. Under display adapters, you should see both the NVIDIA driver and the Microsoft Basic Display Adapter, as shown in Figure 14. Right-click **Microsoft Basic Display Adapter**, and then click **Disable**.

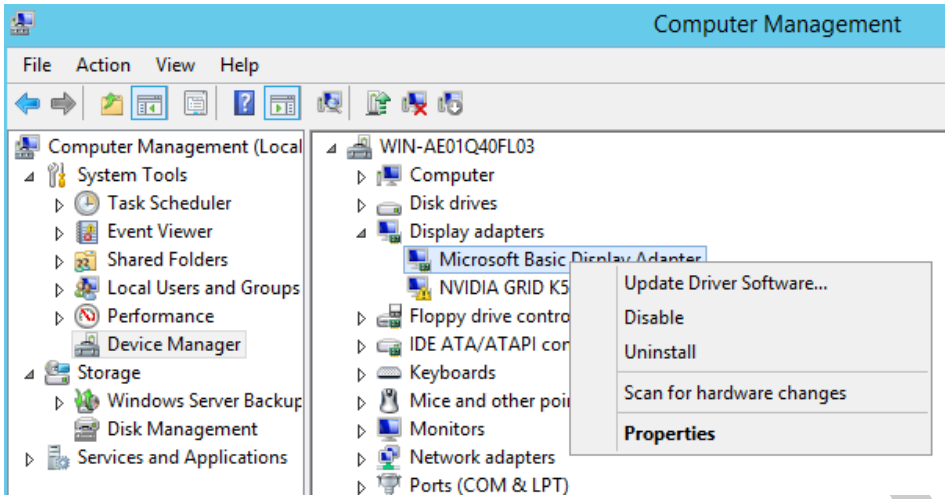

Figure 14: Disabling the Microsoft Basic Display Adapter in Device Manager

13. Now on your local workstation, in AWS Explorer, expand **Amazon EC2**, **Instances**. Right-click your GPU instance, and choose **Stop** (do not choose **Terminate**). This will automatically disconnect your VNC session. Later, you'll use AWS Elastic Beanstalk to start a new instance when you deploy the code.

14. After the instance status changes from stopping to stopped, right-click your GPU instance again in AWS Explorer, and then click **Create Image (EBS AMI)**. Give the image a name and description, and then let it run in the background. There is a small storage charge for the images you save in AWS, but it's convenient to be able to reuse the images with everything pre-installed if you decide to terminate the instance. Whenever you make configuration changes or apply Windows Update on your instance in the future, you should create a new image, and then optionally deregister your older images.

15. After the image is created, look in AWS Explorer under **Amazon EC2**, **AMIs**, and jot down the AMI ID of the image you just created. The ID is case-sensitive.

Now that you have your own AMI, you're ready to switch hats and start working with the code in Visual Studio.

Comparing the Performance of Various Matrix Multiplication Algorithms

Before you deploy the code with AWS Elastic Beanstalk, here's a screenshot of the web application after it completes running. The user interface is simple: it consists of an HTML table listing the timing and relative performance (versus the baseline) of each algorithm, as shown in Figure 15.

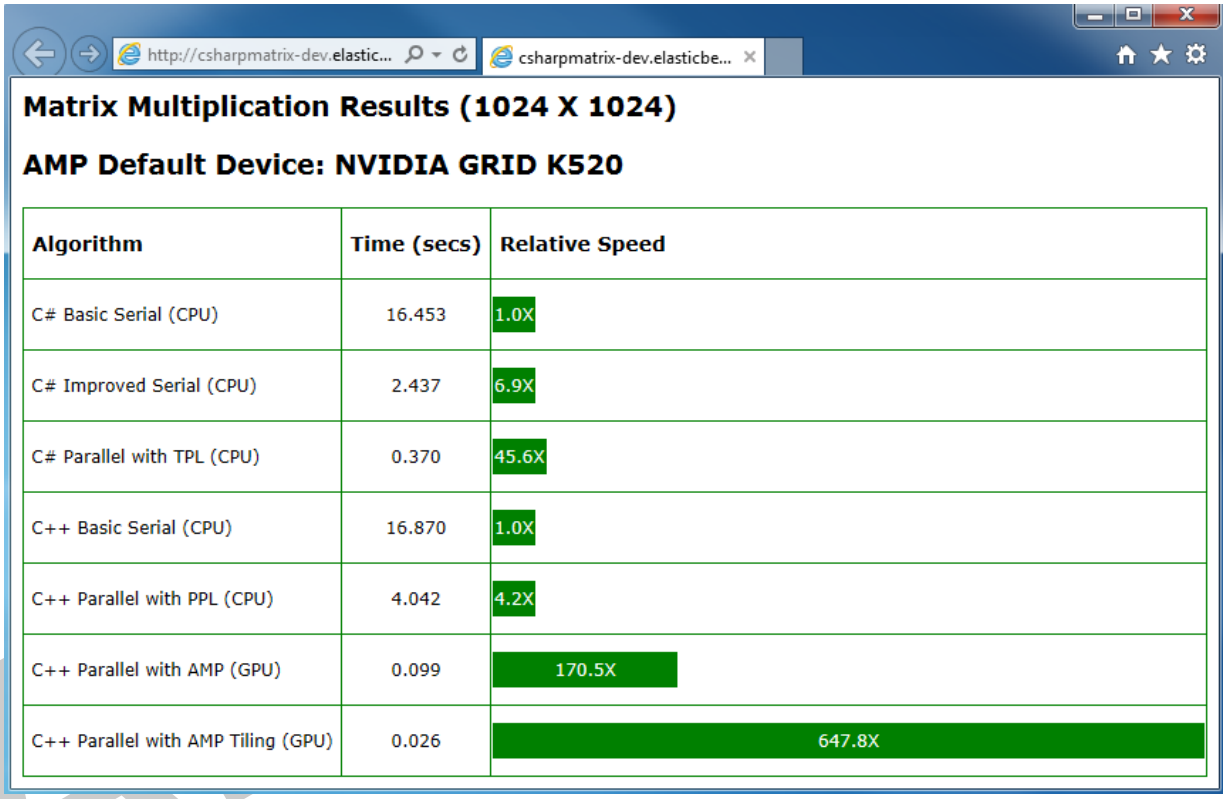

Figure 15: The ASP.NET MVC Application Displaying the Results

You'll notice in the UI that the matrix size used is 1024 x 1024. There are 1,536 CUDA cores on the NVIDIA GPU instance type in Amazon EC2. Because the outer loop of the algorithm will execute in parallel once for each row of the matrix, 1024 was selected as the matrix size to take advantage of a large number of the CUDA cores. Also note that the matrix size must be a multiple of the tile size used in the AMP tiling algorithm.

You may also notice a couple of curiosities in the relative performance of the algorithms. First, the performance of the basic C++ algorithm is almost identical to the performance of the basic C# algorithm. That's interesting, because many

developers suspect that C++ is about twice as fast as C#. A possible explanation for this might be that the C# code is using ragged arrays, which is a known optimization for the .NET Framework.

Another curiosity is that the parallel C# code that uses TPL is about seven times faster than the serial C# code, but the parallel C++ code that uses PPL is only about four times faster than the serial C++ code. Since there are eight virtual cores on the instance, we might expect a parallel algorithm to be about seven times faster. There are ways to get more out of PPL, but that's outside the scope of this paper.

Working with the Code

If you haven't downloaded the Visual Studio solution and source code for this whitepaper yet, you should download it now.[13] Open the CSharpMatrixMultiply solution in Visual Studio. The solution includes two projects. The ASP.NET MVC project is adapted from the basic project that was created with the Visual Studio New Project wizard. The following sections explain the C# code and C++ code in the projects. The C# project has a dependency on the C++ DLL.

Deploying the Web Application with AWS Elastic Beanstalk

To deploy the application by using the image and security group you created earlier:

1. (Recommended) Switch the build configuration in Visual Studio from **Debug** to **Release**.
2. In Solution Explorer, right-click the CSharpMatrixMultiply *project* (**not** the CSharpMatrixMultiply *solution*), and then click **Publish to AWS**.
3. Click **Next** to accept the defaults in the first screen.
4. In the **Application Environment** dialog box, you must provide an environment name, but the default name for this project is too long, so just shorten it until the red border disappears from the text box (Figure 16). Click **Next**.

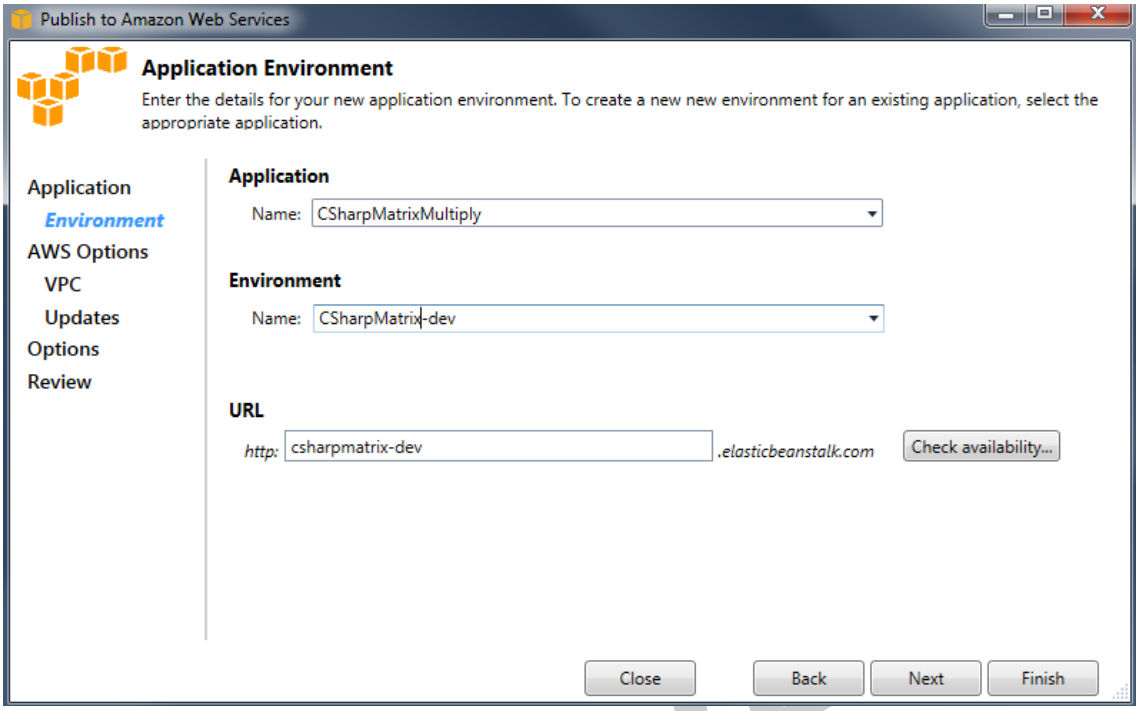

Figure 16: Specifying the Application Environment Details

5. In the **Amazon EC2 Launch Configuration** screen (Figure 17), verify that Windows Server 2012 R2 is selected. For the instance type, select **GPU Double Extra Large**. Select your key pair. Finally, you must provide the AMI ID of the image you created previously. You can find that ID in the Amazon EC2 console under **Images**, or in AWS Explorer, under **Amazon EC2**, **AMIs**. Note that you must enter the ID in lowercase, e.g., ami-12345678. Click **Next**, **Next**, and **Deploy**.

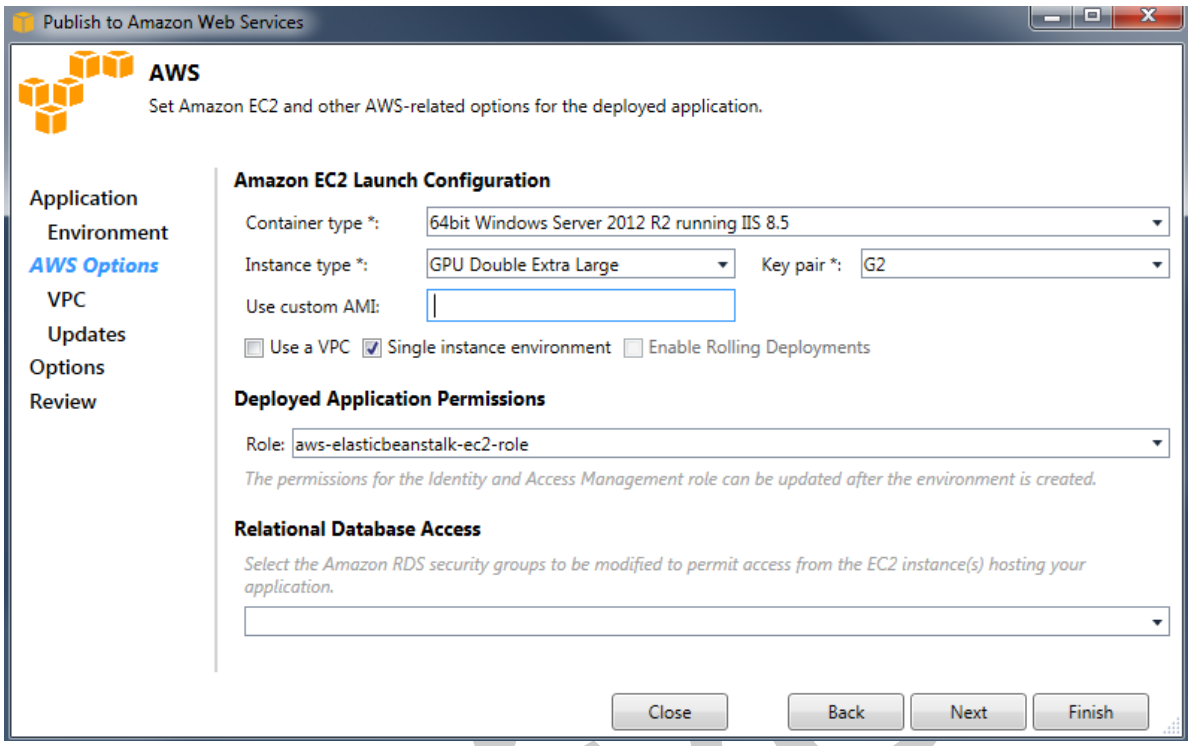

Figure 17: Picking the G2 instance type and Your Custom AMI ID

To ensure smooth builds and deployments of this solution with AWS Elastic Beanstalk, make sure that the version of your AWS Toolkit for Visual Studio is 1.9.2.0 or higher.

The first time you deploy your project with AWS Elastic Beanstalk, it can take 5-10 minutes. When it's done, you may notice that the console or the AWS Toolkit temporarily reports that the deployment is complete but with errors. This can be disconcerting, but if you wait another minute you should see the status change to success.

To run your application, open AWS Explorer and expand the **AWS Elastic Beanstalk** node. Fully expand your environment name and double-click it to see the status pane displayed. The status will show as "Launching" for a few minutes. When the status changes to "Environment is healthy" (again, there could be a delay after it temporarily reports that the environment is unhealthy), click the URL at the top of the status pane. This should launch your default browser, and now you get to wait another couple of minutes while the application performs all seven matrix multiplications in the background. To keep things simple, the web application does not display a progress bar or use an AJAX framework (such as KnockoutJS) for partial updates. (In your production code, you would certainly

want to consider implementing a feature for the user to see the progress of the computation running in the background, and to cancel it if desired.)

After running your application, you may change your program and need to deploy it again. Redeployment is much faster than an initial deployment. In Visual Studio Solution Explorer, right-click the menu for the web project (again, right-click the *project*, **not** the *solution*), and then click **Republish to Environment**.

Using ebextensions with AWS Elastic Beanstalk

When you run the web application, the C++ DLL gets loaded into the IIS process on the web server. This locks the file on the server disk, which can prevent AWS Elastic Beanstalk from being able to overwrite it with a new version when you redeploy your application. One workaround is to connect through VNC and restart the IIS service. Another solution is to use the ebextensions feature that is built into AWS Elastic Beanstalk.

In Solution Explorer, notice the folder in the C# project called .ebextensions (prefaced by a dot). Any text files in this folder that have a file extension of .config will be executed on the server after the deployment. The only tricky thing is that Visual Studio opens .config files in a different editor that doesn't preserve line breaks, so you need to right-click the file and choose **Open With**, **Source Code (Text) Editor**. Here is the file:

```
commands:
  restart-iis:
    command: iisreset /restart
    waitForCompletion:0
```

This ebextensions file instructs AWS Elastic Beanstalk to run the **iisreset** command on the server. For more information, see the blog post "Customizing Windows Elastic Beanstalk Environments," Part 1[14] and Part 2[15], on the AWS .NET Development blog.

Model Code for Data Passed Between Controller and View

The following code is the **Model** class in the web application. In this application, the data flows one way from the Controller to the View.

```
public class TaskResults
{
    public int NumAlgorithms { get; set; }
    public int Dimension { get; set; }
    public string[] Description { get; set; }
    public string[] Time { get; set; }
    public string[] RelativeSpeedLabel { get; set; }
    public int[] PercentOfMax { get; set; }
    public string StatusMessage { get; set; }
    public string AMPDeviceName { get; set; }

    public TaskResults(int _NumAlgorithms)
    {
        NumAlgorithms = _NumAlgorithms;
        Description = new string[NumAlgorithms];
        Time = new string[NumAlgorithms];
        RelativeSpeedLabel = new string[NumAlgorithms];
        PercentOfMax = new int[NumAlgorithms];
        StatusMessage = string.Empty;
        AMPDeviceName = string.Empty;
    }
}
```

Accessing the Model in the View

The following code is the first few lines of the file Index.cshtml. You see that the **TaskResults** object created in the Controller is retrieved through the MVC ViewBag, and then the @ syntax with the Razor Engine is used on the **viewdata** object to insert data (e.g., **@viewdata.AMPDeviceName**) from the Model into HTML.

```
@using CSharpMatrixMultiply.Models;
@{
    ViewBag.Title = "Home Page";
    var viewdata = ViewData["TaskResults"] as TaskResults;
}

<link href="../../Content/MyStyles.css" rel="stylesheet" type="text/css" />

<h3 style="font-family:verdana">Matrix Multiplication Results
    (@viewdata.Dimension X @viewdata.Dimension)</h3>
<h3 style="font-family:verdana">AMP Default Device:
@viewdata.AMPDeviceName</h3>
<h3 style="font-family:verdana; color:red">@viewdata.StatusMessage</h3>
```

Controller Code to Invoke Each Algorithm and Populate the Model

The following code is the main **Controller** class in the web application. It invokes each algorithm (except the first one) three times, calculates the average elapsed time, and stores the results in the **TaskResults** class (the Model).

```
enum Algorithms      // this must exactly duplicate enum in C++
{
    CSharp_Basic = 0,
    CSharp_ImprovedSerial = 1,
    CSharp_TPL = 2,
    CPP_Basic = 3,
    CPP_PPL = 4,
    CPP_AMP = 5,
    CPP_AMPTiling = 6
};

delegate float[][] CSharpMatrixMultiply(float[][] A, float[][] B, int N);

const int TESTLOOPS = 3;
const int N = 1024;          // matrix size must be multiple of C++ tilesize

public unsafe ActionResult Index()
{
    int NumAlgorithms = Enum.GetNames(typeof(Algorithms)).Length;

    var rand = new Random();
    double[] durations = new double[NumAlgorithms];
```

```
    var TaskResults = new TaskResults(NumAlgorithms);
    TaskResults.Description[0] = "C# Basic Serial (CPU)";
    TaskResults.Description[1] = "C# Improved Serial (CPU)";
    TaskResults.Description[2] = "C# Parallel with TPL (CPU)";
    TaskResults.Description[3] = "C++ Basic Serial (CPU)";
    TaskResults.Description[4] = "C++ Parallel with PPL (CPU)";
    TaskResults.Description[5] = "C++ Parallel with AMP (GPU)";
    TaskResults.Description[6] = "C++ Parallel with AMP Tiling (GPU)";
    TaskResults.Dimension = N;
    TaskResults.NumAlgorithms = NumAlgorithms;
    ViewData["TaskResults"] = TaskResults;

    // According to
    // http://www.heatonresearch.com/content/choosing-best-c-array-type-
matrix-multiplication
    // ragged arrays perform better in C# than 2D arrays for matrix
multiplication
    float[][] A = CreateRaggedMatrix(N);
    float[][] B = CreateRaggedMatrix(N);
    FillRaggedMatrix(A, N, rand);
    FillRaggedMatrix(B, N, rand);

    // C++ doesn't need ragged arrays for performance, and it's easier to
marshall
    // and process the data as 2D arrays
    float[,] A2 = new float[N, N];
    float[,] B2 = new float[N, N];
    // for comparing results, use the same random data in C++ as in C#
    CopyRaggedMatrixTo2D(A, A2, N);
    CopyRaggedMatrixTo2D(B, B2, N);

    // warm-up AMP and get GPU name before timing
    var sb = new StringBuilder(256);
    CPPWrapper.WarmUpAMP(sb, sb.Capacity);
    TaskResults.AMPDeviceName = sb.ToString();

    //*** Basic C#. Save this original result for future comparisons.
    long start = DateTime.Now.Ticks;
    float[][] original = MatrixMultiplyBasic(A, B, N);
    long stop = DateTime.Now.Ticks;
    durations[0] = (stop - start) / 10000000.0;

    if (!RunCSharpAlgorithm(
        original,
        A,
        B,
        N,
        MatrixMultiplySerial,
        "C# Improved Serial",
        (int)Algorithms.CSharp_ImprovedSerial,
        TaskResults,
        ref durations))
    {
        return PartialView();
```

```
    }

    if (!RunCSharpAlgorithm(
        original,
        A,
        B,
        N,
        MatrixMultiplyTPL,
        "C# TPL",
        (int)Algorithms.CSharp_TPL,
        TaskResults,
        ref durations))
    {
        return PartialView();
    }

    if (!RunCPPAlgorithm(original, A2, B2, N, "C++ Basic",
        (int)Algorithms.CPP_Basic, TaskResults, ref durations))
    {
        return PartialView();
    }

    if (!RunCPPAlgorithm(original, A2, B2, N, "C++ PPL",
        (int)Algorithms.CPP_PPL, TaskResults, ref durations))
    {
        return PartialView();
    }

    if (!RunCPPAlgorithm(original, A2, B2, N, "C++ AMP",
        (int)Algorithms.CPP_AMP, TaskResults, ref durations))
    {
        return PartialView();
    }

    if (!RunCPPAlgorithm(original, A2, B2, N, "C++ AMP Tiling",
        (int)Algorithms.CPP_AMPTiling, TaskResults, ref durations))
    {
        return PartialView();
    }

    var slowest = durations.Max();
    var fastest = durations.Min();

    // populate the Model for the HTML table in the View
    for (int k = 0; k < NumAlgorithms; k++)
    {
        TaskResults.Time[k] = string.Format("{0:0.000}", durations[k]);
        TaskResults.RelativeSpeedLabel[k] = string.Format("{0:0.0}X", slowest
/ durations[k]);
        TaskResults.PercentOfMax[k] = (int)(fastest / durations[k] * 100.0);
    }

    return PartialView();
}
```

```
bool RunCSharpAlgorithm(
        float[][] original,
        float[][] A,
        float[][] B,
        int N,
        CSharpMatrixMultiply function,
        string FunctionName,
        int AlgorithmIndex,
        TaskResults results,
        ref double[] durations)
{
    double[] test_durations = new double[TESTLOOPS];

    for (int k = 0; k < TESTLOOPS; k++)
    {
        long start = DateTime.Now.Ticks;
        float[][] C = function(A, B, N);
        test durations[k] = (DateTime.Now.Ticks - start) / 10000000.0;

        if (!CompareMatrixes(original, C, N))
        {
            results.StatusMessage = "Error verifying " + FunctionName;
            return false;
        }
    }

    durations[AlgorithmIndex] = test_durations.Average();

    return true;
}

unsafe bool RunCPPAlgorithm(
    float[][] original,
    float[,] A2,
    float[,] B2,
    int N,
    string FunctionName,
    int AlgorithmIndex,
    TaskResults results,
    ref double[] durations)
{
    double[] test_durations = new double[TESTLOOPS];

    for (int k = 0; k < TESTLOOPS; k++)
    {
        // allocate memory in C# to simplify marshalling/deallocation
        float[,] C2 = new float[N, N];

        long start = DateTime.Now.Ticks;
        fixed (float* pA2 = &A2[0, 0])
        fixed (float* pB2 = &B2[0, 0])
        fixed (float* pC2 = &C2[0, 0])
        {
            var error = new StringBuilder(1024);   // allocate string memory
```

```
            if (!CPPWrapper.CallCPPMatrixMultiply(AlgorithmIndex,
                pA2, pB2, pC2, N, error, error.Capacity))
            {
                results.StatusMessage = error.ToString();
                return false;
            }
        }

        if (!CompareRaggedMatrixTo2D(original, C2, N))
        {
            results.StatusMessage = "Error verifying " + FunctionName;
            return false;
        }

        test_durations[k] = (DateTime.Now.Ticks - start) / 10000000.0;
    }

    durations[AlgorithmIndex] = test durations.Average();
    return true;
}

// Standard algorithm
float[][] MatrixMultiplyBasic(float[][] A, float[][] B, int N)
{
    float[][] C = CreateRaggedMatrix(N);   // C is the result matrix

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                C[i][j] += A[i][k] * B[k][j];

    return C;
}

// This function was developed by Heaton Research and is licensed under the
Apache License, Version 2.0,
// available here: https://www.apache.org/licenses/LICENSE-2.0.html
// Improve the basic serial algorithm with optimized index order
float[][] MatrixMultiplySerial(float[][] A, float[][] B, int N)
{
    float[][] C = CreateRaggedMatrix(N);

    // according to http://www.heatonresearch.com/content/choosing-best-c-
array-type-matrix-multiplication
    // this ikj index order performs the best for C# matrix multiplication
    for (int i = 0; i < N; i++)
    {
        float[] arowi = A[i];
        float[] crowi = C[i];
        for (int k = 0; k < N; k++)
        {
            float[] browk = B[k];
            float aik = arowi[k];
            for (int j = 0; j < N; j++)
            {
```

```
                crowi[j] += aik * browk[j];
            }
        }
    }

    return C;
}

// Parallel algorithm using TPL
float[][] MatrixMultiplyTPL(float[][] A, float[][] B, int N)
{
    float[][] C = CreateRaggedMatrix(N);

    Parallel.For(0, N, i =>
    {
        float[] arowi = A[i];
        float[] crowi = C[i];
        for (int k = 0; k < N; k++)
        {
            float[] browk = B[k];
            float aik = arowi[k];
            for (int j = 0; j < N; j++)
            {
                crowi[j] += aik * browk[j];
            }
        }
    });

    return C;
}
```

C# Basic Serial (CPU)

A basic algorithm for matrix multiplication is used as the baseline for the algorithms in subsequent sections. There is only one optimization applied in this basic algorithm. When using two-dimensional arrays in the .NET Framework, method calls would ordinarily be made to the **Array** class. Since the inner loop executes so many times, that's expensive. But there is a simple workaround: use ragged arrays. For example, instead of declaring a 10x20 array like this:

```
double[,] MyArray = new double[10,20];
```

declare it like this, and create each row as a separate array of 20 columns in a **for** loop:

```
double[][] MyArray = new double[10][];
for (int i = 0; i < 10; i++)
```

```
        MyArray[i] = new double[20];
```

Here is the code for basic matrix multiplication. This will execute in serial fashion on the CPU:

```
float[][] MatrixMultiplyBasic(float[][] A, float[][] B, int N)
{
    float[][] C = CreateRaggedMatrix(N);   // C is the result matrix

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                C[i][j] += A[i][k] * B[k][j];

    return C;
}
```

C# Optimized Serial (CPU)

The code for this algorithm was obtained from the article Choosing the Best C# Array Type for Matrix Multiplication[16] By Heaton Research. In the article, the author writes several variations of the order of the **for** loop indexes, and measures the timing of each. For this whitepaper, we are using the variation that was found to perform the best with the .NET Framework 4.5.

```
float[][] MatrixMultiplySerial(float[][] A, float[][] B, int N)
{
    float[][] C = CreateRaggedMatrix(N);

    // according to http://www.heatonresearch.com/content/choosing-best-c-
array-type-matrix-multiplication
    // this ikj index order performs the best for C# matrix multiplication
    for (int i = 0; i < N; i++)
    {
        float[] arowi = A[i];
        float[] crowi = C[i];
        for (int k = 0; k < N; k++)
        {
            float[] browk = B[k];
            float aik = arowi[k];
            for (int j = 0; j < N; j++)
```

```
            {
                crowi[j] += aik * browk[j];
            }
        }
    }

    return C;
}
```

C# Parallel with TPL (CPU)

The following code simply replaces the standard outer loop in the previous algorithm with a **Parallel.For** loop from the .NET Framework Task Parallel Library (TPL). For more information, see Matrix Multiplication in Parallel with C# and the TPL[17] by James D. McCaffrey.

```
float[][] MatrixMultiplyTPL(float[][] A, float[][] B, int N)
{
    float[][] C = CreateRaggedMatrix(N);

    Parallel.For(0, N, i =>
    {
        float[] arowi = A[i];
        float[] crowi = C[i];
        for (int k = 0; k < N; k++)
        {
            float[] browk = B[k];
            float aik = arowi[k];
            for (int j = 0; j < N; j++)
            {
                crowi[j] += aik * browk[j];
            }
        }
    });

    return C;
}
```

C++ Basic Serial (CPU)

If you decide to build your own program, you must follow the steps in the blog post How to use C++ AMP from C#[18] on the Parallel Programming with .NET

blog on MSDN. If you only want to download and run the sample code provided with this whitepaper, there is no need to follow that procedure, because those steps have already been included in the Visual Studio solution.

One difference between our solution and the information in the blog post is that our solution uses all 64-bit code. When combining C# and C++, you need to be careful to use the same platform in each language. The platform is usually set to **Any CPU** in C#, but it must be changed to **x64** in the Visual Studio Configuration Manager, as shown in Figure 18.

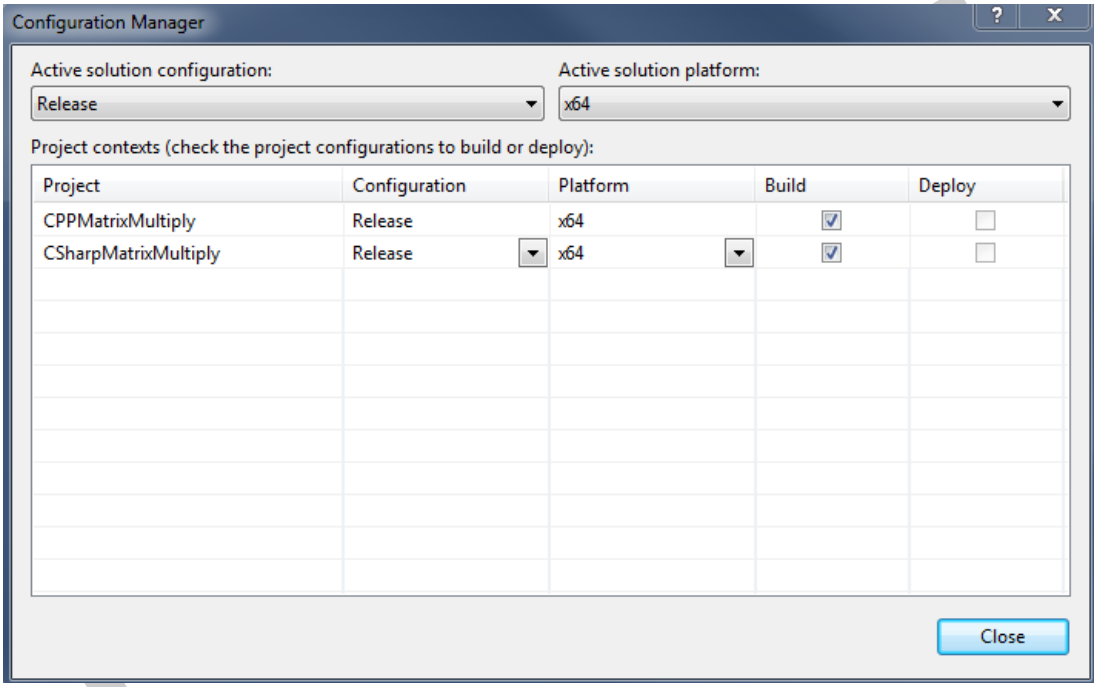

Figure 18: Setting the Platform to x64 in the Visual Studio Configuration Manager

See the blog post [Debugging VS2013 websites using 64-bit IIS Express](#)[19] for additional helpful information.

Before you can invoke C++ functions from C#, you need to declare them for P/Invoke on the C# side. The following code shows the **CPPWrapper** class in the Controller folder in the Visual Studio solution. As required, these methods are declared with the unsafe keyword in C#. Rather than create a public entry point for each C++ algorithm, it was deemed a bit cleaner to create a single function to call each one based on the algorithm index passed in. This simplifies the exception handling which had to be written in C++. I would have liked to write a single exception handler in C# for all the calls to the different algorithms,

including C++, but it was necessary to write an error handler in C++ for the error codes that can be returned by C++ AMP.

```
public class CPPWrapper
{
    [DllImport("CPPMatrixMultiply.dll",
        CallingConvention = CallingConvention.StdCall,
        CharSet = CharSet.Unicode)]
    public extern unsafe static bool
    CallCPPMatrixMultiply(int algorithm, float* A, float* B, float* C,
        int N, StringBuilder error, int errsize);

    [DllImport("CPPMatrixMultiply.dll",
        CallingConvention = CallingConvention.StdCall,
        CharSet = CharSet.Unicode)]
    public extern unsafe static void WarmUpAMP(StringBuilder buffer, int
bufsize);
}
```

Here is the C++ dispatcher function, which is exported for C#:

```
extern "C" __declspec (dllexport) bool _stdcall
CallCPPMatrixMultiply(int algorithm, float A[], float B[], float C[],
  int N, wchar_t* error, size_t errsize)
{
  try
  {
  switch (algorithm)
  {
  case Algorithms::CPP_Basic:
     MatrixMultiplyBasic(A, B, C, N); break;
  case Algorithms::CPP_PPL:
     MatrixMultiplyPPL(A, B, C, N); break;
  case Algorithms::CPP_AMP:
     MatrixMultiplyAMP(A, B, C, N); break;
  case Algorithms::CPP_AMPTiling:
     MatrixMultiplyTiling(A, B, C, N); break;
  default:
     wcscpy_s(error, errsize, L"Invalid C++ algorithm index.");
     return false;
  }
  }
  catch (concurrency::runtime_exception& ex)
```

```
    {
        std::wstring result = stows(ex.what());
        wcscpy_s(error, errsize, result.c_str());
        return false;
    }

    return true;
}
```

Now that you've taken care of those preliminaries, you're ready to implement the C++ function for basic matrix multiplication. It looks very similar to the basic algorithm in C# except that it doesn't use ragged arrays, and it introduces a temporary `sum` variable to reduce array references to the result array in the inner loop.

```
void MatrixMultiplyBasic(float A[], float B[], float C[], int N)
{
    for (int i = 0; i < N; i++)
    {
      for (int j = 0; j < N; j++)
      {
          float sum = 0.0;
          for (int k = 0; k < N; k++)
          {
             sum += A[i*N + k] * B[k*N + j];
          }
          C[i*N + j] = sum;
      }
    }
}
```

C++ Parallel with PPL (CPU)

The next optimization is to rewrite the serial C++ function as a parallel function. This code will still be running on the CPU, but it will give us an interesting comparison with the parallel code we'll write later to run on the GPU.

In the past, writing parallel code in Windows with the Win32 thread APIs was complicated. There are still many difficulties in multithreaded programming, but

now the Microsoft Parallel Patterns Library (PPL) makes it much easier. For more information about PPL, see the following:

- This article in the MSDN Library explains a parallel matrix multiplication algorithm written in C++ using PPL: How to: Write a parallel_for Loop[20]
- This article describes several optimization techniques for writing parallel for loops in C++: C++11: Multi-core Programming – PPL Parallel Aggregation Explained.[21]

Here's the non-optimized parallel C++ function:

```
void MatrixMultiplyPPL(float A[], float B[], float C[], int N)
{
  parallel_for(0, N, [&](int i)
  {
    for (int j = 0; j < N; j++)
    {
        float sum = 0.0;
        for (int k = 0; k < N; k++)
        {
              sum += A[i*N + k] * B[k*N + j];
        }
        C[i*N + j] = sum;
     }
  });
}
```

C++ Parallel with AMP (GPU)

Now you're ready to write AMP code. To get started, you may want to review the blog post How to measure the performance of C++ AMP algorithms[22] on the Parallel Programming in Native Code blog on MSDN. As that author points out, there is overhead when AMP initializes itself on first use. It enumerates the GPU devices in the system and picks the default one. The idea of warming up AMP before timing it may or may not apply to your use case, but the code provided with this whitepaper does implement such a function. The following function returns the name of the GPU device so it can be displayed in the ASP.NET MVC web page.

```
// Return the name of the default GPU device (or the emulator if no GPU
exists).
```

```
// AMP will enumerate devices to initialize itself outside of the timing
code.
extern "C" __declspec (dllexport) void _stdcall
WarmUpAMP(wchar_t* buffer, size_t bufsize)
{
    accelerator default_device;
    wcscpy_s(buffer, bufsize, default_device.get_description().c_str());
}
```

String types in C# and C++ are not directly compatible, but there are various ways to pass strings between them (this is called *marshaling*). In all cases, it's important to pay attention to where the string memory is allocated and how it will be freed. The P/Invoke declaration in C# must be carefully written to match the string-passing technique you decide to use in C++. The technique used in the previous code is to allocate a **StringBuilder** object with a fixed capacity in C# before passing it into C++. That way, the C# side is responsible for freeing the memory when the object goes out of scope, which only happens after the C++ function is done writing to the memory. The C++ code just copies the name of the GPU device into the buffer passed in from C#.

The next task is to adapt the parallel C++ matrix multiplication algorithm to use AMP. The following AMP code is based on the Matrix Multiplication Sample[23] on the Parallel Programming in Native Code blog on MSDN.

```
void MatrixMultiplyAMP(float A[], float B[], float C[], int N)
{
  extent<2> e_a(N, N), e_b(N, N), e_c(N, N);

  array_view<float, 2> a(e_a, A);
  array_view<float, 2> b(e_b, B);
  array_view<float, 2> c(e_c, C);
  c.discard_data();       // avoid copying memory to GPU

  parallel_for_each(c.extent, [=](index<2> idx) restrict(amp)
  {
  int row = idx[0];
  int col = idx[1];
  float sum = 0;
  for (int inner = 0; inner < N; inner++)
  {
```

```
    index<2> idx_a(idx[0], inner);
    index<2> idx_b(inner, idx[1]);
    sum += a[idx_a] * b[idx_b];
    }
    c[idx] = sum;
    });

    c.synchronize();
}
```

C++ Parallel with AMP Tiling (GPU)

Finally, let's take another step with the AMP code to use a technique called *tiling*. In a nutshell, tiling is a method of optimizing the way the algorithm uses memory in the GPU. When you call C++ AMP from C#, there are four levels of memory you should be aware of:

- **Managed memory.** This lives in RAM associated with the CPU and the .NET Framework CLR managed process, and is controlled by the .NET Framework garbage collector. Data passed between C# and C++ must be "marshaled" between managed and unmanaged memory according to very particular rules, such as padding.
- **Unmanaged memory.** This also lives in RAM associated with the CPU, but this memory space requires Win32 memory APIs and does not include a garbage collector.
- **Global memory on the GPU.** Programming in AMP requires that data be moved—with thread synchronization—between unmanaged memory and the GPU.
- **Registers associated with each thread on the GPU.** Accessing data in these registers can be 1000 times faster than GPU global memory, so the idea is to move frequently accessed data into the registers. But the registers aren't large enough to hold an entire matrix, so algorithms must be written to process one "tile" at a time and then move another tile into the registers, and so on. A full explanation of tiling is beyond the scope of this whitepaper, but [this article by Daniel Moth covers it well](#).[24]

Here is the C++ AMP code with a tiling algorithm:

```
const int TILESIZE = 8;  // array size passed in must be a multiple of
TILESIZE

void MatrixMultiplyTiling(float A[], float B[], float C[], int N)
{
   assert((N % TILESIZE) == 0);
   array_view<const float, 2> a(N, N, A);
   array_view<const float, 2> b(N, N, B);
   array_view<float, 2> c(N, N, C);
   c.discard_data();

   parallel_for_each(c.extent.tile<TILESIZE, TILESIZE>(),
   [=](tiled_index<TILESIZE, TILESIZE> t_idx) restrict(amp)
   {
   int row = t_idx.local[0];
   int col = t_idx.local[1];
   tile_static float locA[TILESIZE][TILESIZE];
   tile_static float locB[TILESIZE][TILESIZE];
   float sum = 0;
   for (int i = 0; i < a.extent[1]; i += TILESIZE)
   {
   locA[row][col] = a(t_idx.global[0], col + i);
   locB[row][col] = b(row + i, t_idx.global[1]);
   t_idx.barrier.wait();

   for (int k = 0; k < TILESIZE; k++)
   sum += locA[row][k] * locB[k][col];
   t_idx.barrier.wait();

   }
   c[t_idx.global] = sum;
   });
   c.synchronize();
}
```

Conclusion

This whitepaper demonstrated how to set up the G2 instance type in Amazon EC2 with Windows Server. The NVIDIA GPU on those instances provides 1,536 cores that developers can use for compute-intensive application functions. But programming the GPU requires the C or C++ language, whereas most Windows developers are using C#. This article showed how to pass data between C# and

C++, and how to use the C++ AMP library to make GPU programming accessible and highly productive for C# web developers on Windows.

The tiled matrix multiplication algorithm written in C++ AMP was hundreds of times faster than the basic algorithm written in C#.

Further Reading

- AWS Toolkit for Visual Studio[25]
- AWS for Windows and .NET Developer Center[26]
- Getting Started with Amazon EC2 Windows Instances[27]
- Elastic Beanstalk Documentation[28]
- C++ AMP documentation[29]
- ASP.NET MVC documentation[30]

Notes

[1] http://do.awsstatic.com/whitepapers/CSharpMatrixMultiply.zip

[2] https://bitbucket.org/multicoreware/cppamp-driver-ng/overview

[3] https://ampalgorithms.codeplex.com/documentation

[4] http://blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx

[5] http://aws.amazon.com/ec2/instance-types/

[6] http://www.nvidia.com/object/cuda_home_new.html

[7] http://aws.amazon.com/visualstudio/

[8] http://aws.amazon.com/free/

[9] http://www.microsoft.com/en-us/download/details.aspx?id=40784

[10] http://www.realvnc.com/

[11] http://www.realvnc.com/

[12] http://www.NVIDIA.com/download/driverResults.aspx/74642/en-us

[13] http://do.awsstatic.com/whitepapers/CSharpMatrixMultiply.zip

[14] http://blogs.aws.amazon.com/net/post/Tx1RLX98N5ERPSA/Customizing-Windows-Elastic-Beanstalk-Environments-Part-1

[15] http://blogs.aws.amazon.com/net/post/Tx2EMAYCXUW3HAK/Customizing-Windows-Elastic-Beanstalk-Environments-Part-2

[16] http://www.heatonresearch.com/content/choosing-best-c-array-type-matrix-multiplication

[17] http://jamesmccaffrey.wordpress.com/2012/04/22/matrix-multiplication-in-parallel-with-c-and-the-tpl/

[18] http://blogs.msdn.com/b/pfxteam/archive/2011/09/21/10214538.aspx

[19] http://blogs.msdn.com/b/rob/archive/2013/11/14/debugging-vs2013-websites-using-64-bit-iis-express.aspx

[20] http://msdn.microsoft.com/en-us/library/dd728073.aspx

[21] https://katyscode.wordpress.com/2013/08/17/c11-multi-core-programming-ppl-parallel-aggregation-explained/

[22] http://blogs.msdn.com/b/nativeconcurrency/archive/2011/12/28/how-to-measure-the-performance-of-c-amp-algorithms.aspx

[23] http://blogs.msdn.com/b/nativeconcurrency/archive/2011/11/02/matrix-multiplication-sample.aspx

[24] http://msdn.microsoft.com/en-us/magazine/hh882447.aspx

[25] http://aws.amazon.com/visualstudio/

[26] http://aws.amazon.com/net/

[27] http://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/EC2Win_GetStarted.html

[28] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/customize-containers-windows-ec2.html

[29] http://msdn.microsoft.com/en-us/library/hh265137.aspx

[30] http://www.asp.net/mvc